TEMA:

Recursividad

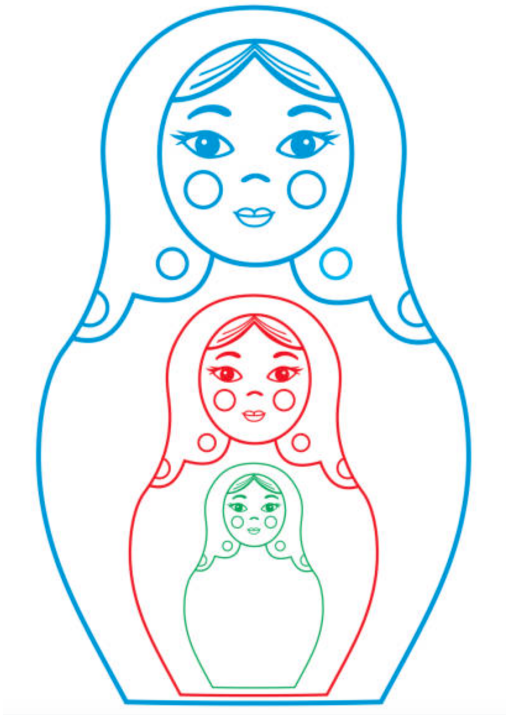
Emma Rollón
erollon@cs.upc.edu

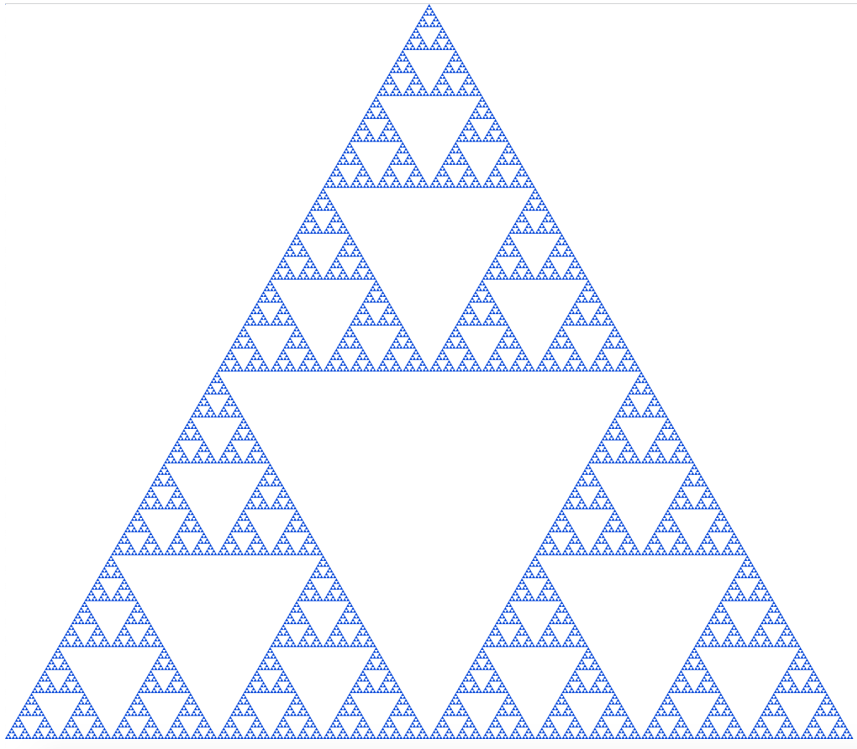
Departamento de Ciencias de la Computación

Índice

Introducción

Recursividad es la forma en la que se especifica un proceso basado en su propia definición.





Principio: reducir un problema complejo en una instancia más sencilla del mismo problema.

Introducción

Recursividad e inducción matemática están íntimamente relacionados.

La suma de los n primeros números impares es n^2. Es decir:

$$\sum_{i=1}^{n}(2i-1) = n^2$$

- Caso base: $n = 1 \Rightarrow 2 * 1 - 1 = 1^2$.
- Paso de inducción: asumiendo que se cumple hasta n, vamos a demostrar $n+1$:

$$\begin{aligned}
\sum_{i=1}^{n+1}(2i-1) &= \sum_{i=1}^{n}(2i-1) + 2(n+1) - 1 \\
&= n^2 + 2n + 2 - 1 \\
&= n^2 + 2n + 1 \\
&= (n+1)^2
\end{aligned}$$

Introducción

Ejemplo matemático (factorial):

$$n! = \begin{cases} n \times (n-1) \times \ldots \times 2 \times 1 & , n > 0 \\ 1 & , n = 0 \end{cases} \qquad n! = \begin{cases} n \times (n-1)! & , n > 0 \\ 1 & , n = 0 \end{cases}$$

En nuestros códigos:

```
int mifunc(...) {
    ...
    ... mifunc(...);
    ...
}
```

```
void miproc(...) {
    ...
    miproc(...);
    ...
}
```

Características

Para que una definición recursiva sea correcta, tiene que cumplir:

1. Uno o más casos base
2. Uno o más casos recursivos:
   - respecto un problema "más pequeño"
   - llega al caso base

} Exhaustivos

Ejemplos:

$$n! = \begin{cases} n \times (n-1)! & , n > 0 \\ 1 & , n = 0 \end{cases}$$

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & , n > 1 \\ 1 & , 0 <= n <= 1 \end{cases}$$

¿Cómo se implementa? (I)

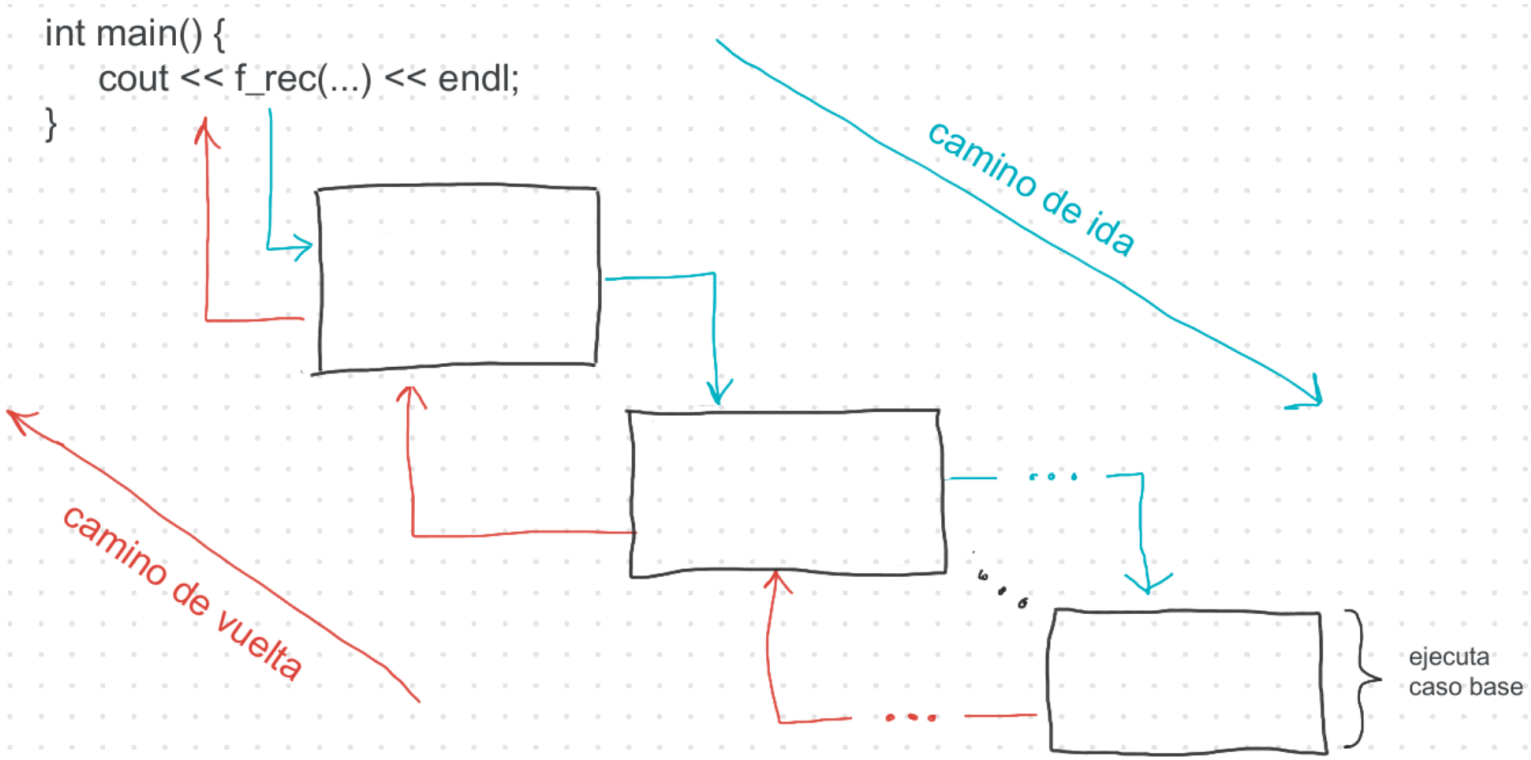
Si tenemos la definición:

$$n! = \begin{cases} n \times (n-1)! & n > 0 \\ 1 & n = 0 \end{cases}$$

La traducción es directa:

```
// Pre: n >= 0
// Post: retorna el factorial de n
int factorial(int n) {
    if (n == 0) return 1;
    else return n*factorial(n - 1);
}
```

¿Cómo se ejecuta?



¡Cuidado!

Si no llega a algún caso base:

- → Entra en bucle infinito (es decir, no hace camino de vuelta).
- → Al ejecutarlo, tendrás un error de "Segmentation fault".

Tres observaciones:

1. Todo lo que se puede implementar en iterativo se puede implementar en recursivo.

2. No todo lo recursivo se puede implementar en iterativo sin almacenamiento extra.

3. En general, las soluciones recursivas son más simples que las iterativas. Pero:

   $\Rightarrow$ la complejidad se oculta en la ejecución

   $\Rightarrow$ son un poquito más ineficientes

   $\Rightarrow$ algunas muy naturales son extremadamente ineficientes

¿Cómo se implementa? (II)

Si no tenemos la definición (la tenemos que pensar nosotros):

1. Especificar qué se tiene que hacer (Pre/Post)
2. Caso recursivo:
   - poner un ejemplo concreto del problema a solucionar
   - suponer que la función/acción está implementada
   - utilizar (llamar a) esa función/acción para resolver un problema un poco más pequeño que el del ejemplo original (pero relacionado)
   - usar ese resultado para solucionar el ejemplo original
3. Caso base:
   - pensar qué problema original es muy fácil de resolver (sin necesidad de hacer ninguna llamada recursiva) ...
   - ... sabiendo que una cascada de llamadas recursivas (sobre los problemas complejos) siempre llegan a él.

Observación:

Al hacer el razonamiento, los pasos 2 y 3 se pueden intercambiar de orden.

Ejemplos a hacer

- Calcular el número de fibonacci de orden n, para todas las $n >= 0$
- Calcular el número de dígitos de un natural $n >= 0$
- Retornar true si un número natural es par, false en caso contrario. No se puede utilizar la operación %.
- Escribir la representación en binario de un natural $n >= 0$.
- Retornar el número de letras 'a' que hay en una secuencia de caracteres acabada en '.'.
- Calcular la potencia x^y de dos enteros positivos x, y.
- Retornar true si un natural n contiene un dígito d, false en caso contrario.
- Retornar true si un natural n está formado por dígitos iguales.

Soluciones

```
// Pre: n >= 0
// Post: retorna el número de fibonacci de orden n
int fib(int n) {
    if (n < 2) return 1;
    else return fib(n - 1) + fib(n - 2);
}
```

```
// Pre: n >= 0
// Post: retorna el número de dígitos de n
int ndigitos(int n) {
    if (n < 10) return 1;
    else return 1 + ndigitos(n/10);
}
```

```
// Pre: n >= 0
// Post: retorna true si n es par, false en caso contrario
bool par(int n) {
    if (n == 0) return true;
    else if (n == 1) return false;
    else return par(n - 2);
}
```

Soluciones

```
// Pre: n >= 0
// Post: escribe la representación en binario de n
void binario(int n) {
    if (n < 2) cout << n;          // alternativa:
    else {                         // if (n > 1) binario(n/2);
        binario(n/2);              // cout << n%2;
        cout << n%2;
    }
}
```

```
// Pre: (implícito: secuencia de caracteres acabada en '.')
// Post: retorna el número de 'a' en la secuencia
int cuantas_a() {
    char c;
    cin >> c;
    if (c == '.') return 0;
    else if (c == 'a') return 1 + cuantas_a();
    else return cuantas_a();
}
```

Soluciones

```
// Pre: x >= 0, y >= 0
// Post: retorna x elevado a y (0 elevado a 0 es 1)
int potencia(int x, int y) {
    if (y == 0) return 1;
    else return x*potencia(x, y - 1);
}
```

```
// Pre: n >= 0, 0 <= d <= 9
// Post: retorna true si d es un dígito de n, false si no
bool contiene(int n, int d) {
    if (n < 10) return n == d;
    else return n%10 == d or contiene(n/10, d);
}
```

```
// Pre: n >= 0
// Post: retorna true si n está formado por un único dígito,
//         false si no
bool iguales(int n) {
    if (n < 10) return true;
    else return n%10 == (n/10)%10 and iguales(n/10);
}
```